UNIVERSITY OF MICHIGAN

EECS 373
Midterm 1 Review

Rajin & Guthrie

Winter 24

Topics

- Embedded system definition and market
- Technology trends
- Embedded applications
- ARM architecture, assembly, and ABI
- MMIO
- Debugging
- APB
- Build process
- Aspects of ANSI C related to embedded systems
- Interrupts
- Timers

Embedded Systems and Market Values

- Embedded System
  - An (application-specific) computer within something else that is not generally regarded as a computer.
- Common Requirements
  - Timely (hard real-time), Wireless
  - Reliable
  - First time correct
  - Rapidly implemented
  - Low price
  - High performance
  - Low power
  - Embodying deep domain knowledge
  - Beautiful

Technology Trends

Technology Scaling

- Moore's Law
  - Made transistors cheap
- Dennard Scaling
  - Made them fast
  - But power density undermines

Result

- Fixed transistor count
  - Exponentially lower cost
  - Exponentially lower power
- Small, cheap, and low-power
  - Microcontrollers
  - Sensors
  - Memory

Technology Innovations

- MEMS technology
  - Micro-fabricated sensors
- New memories
  - New cell structures (11T)
  - New tech (FeRAM, FinFET)
- Near-threshold computing
  - Minimize active power
  - Minimize static power
- New wireless systems
  - Radio architectures
  - Modulation schemes
- Energy harvesting

ARMv7-M Architecture

- Comprised of both 16 and 32 bit instructions
  - 16 bit instructions have access only to “low registers” (r0-r7)
  - 32 bit instructions often have more functionality than 16 bit instructions

- Most instructions we use fall in the following categories:
  - Data Processing
  - Load/Store
  - Branch
  - Miscellaneous

Data Processing Assembly Instructions

Table A4-2 Standard data-processing instructions

Mnemonic	Instruction	Notes
ADC	Add with Carry	-
ADD	Add	Thumb permits use of a modified immediate constant or a zero-extended 12-bit immediate constant.
ADR	Form PC-relative Address	First operand is the PC. Second operand is an immediate constant. Thumb supports a zero-extended 12-bit immediate constant. Operation is an addition or a subtraction.
AND	Bitwise AND	-
BIC	Bitwise Bit Clear	-
CMN	Compare Negative	Sets flags. Like ADD but with no destination register.
CMP	Compare	Sets flags. Like SUB but with no destination register.
EOR	Bitwise Exclusive OR	-
MOV	Copies operand to destination	Has only one operand, with the same options as the second operand in most of these instructions. If the operand is a shifted register, the instruction is an LSL, LSR, ASR, or ROR instruction instead. See *Shift instructions* on page A4-10 for details. Thumb permits use of a modified immediate constant or a zero-extended 16-bit immediate constant.
MVN	Bitwise NOT	Has only one operand, with the same options as the second operand in most of these instructions.

Load/Store Assembly Instructions

Table A4-10 Load and store instructions

Data type	Load	Store	Load unprivileged	Store unprivileged	Load exclusive	Store exclusive
32-bit word	LDR	STR	LDRT	STRT	LDREX	STREX
16-bit halfword	-	STRH	-	STRHT	-	STREXH
16-bit unsigned halfword	LDRH	-	LDRHT	-	LDREXH	-
16-bit signed halfword	LDRSH	-	LDRSHT	-	-	-
8-bit byte	-	STRB	-	STRBT	-	STREXB
8-bit unsigned byte	LDRB	-	LDRBT	-	LDREXB	-
8-bit signed byte	LDRSB	-	LDRSBT	-	-	-
two 32-bit words	LDRD	STRD	-	-	-	-

Branch Assembly Instructions

Table A4-1 Branch instructions

Instruction	Usage	Range
B on page A6-40	Branch to target address	+/–1 MB
CBNZ, CBZ on page A6-52	Compare and Branch on Nonzero, Compare and Branch on Zero	0-126 B
BL on page A6-49	Call a subroutine	+/–16 MB
BLX (register) on page A6-50	Call a subroutine, optionally change instruction set	Any
BX on page A6-51	Branch to target address, change instruction set	Any
TBB, TBH on page A6-258	Table Branch (byte offsets)	0-510 B
	Table Branch (halfword offsets)	0-131070 B

Code	Meaning (for `cmp` or `subs`)	Flags Tested
`eq`	Equal.	`Z==1`
`ne`	Not equal.	`Z==0`
`cs` or `hs`	Unsigned higher or same (or carry set).	`C==1`
`cc` or `lo`	Unsigned lower (or carry clear).	`C==0`
`mi`	Negative. The mnemonic stands for "minus".	`N==1`
`pl`	Positive or zero. The mnemonic stands for "plus".	`N==0`
`vs`	Signed overflow. The mnemonic stands for "V set".	`V==1`
`vc`	No signed overflow. The mnemonic stands for "V clear".	`V==0`
`hi`	Unsigned higher.	`(C==1) && (Z==0)`
`ls`	Unsigned lower or same.	`(C==0) || (Z==1)`
`ge`	Signed greater than or equal.	`N==V`
`lt`	Signed less than.	`N!=V`
`gt`	Signed greater than.	`(Z==0) && (N==V)`
`le`	Signed less than or equal.	`(Z==1) || (N!=V)`
`al` (or omitted)	Always executed.	None tested.

Application Binary Interface (ABI)

Detailed version

- Pass: r0-r3 (Caller saved)
- Return: r0 or r0-r1

Callee saved variables: r4-r8, r11, maybe r9, r10

- Static base: r9 (might offset from this to write)
- Stack limit checking: r10 (SP >= r10)
- Veneers, scratch: r12 (lillypad)
- Stack pointer: r13
- Link register (function call return address): r14
- Program counter: r15

Simple version

- Callee preserves r4-r11 and r13
- Caller preserves r0-r3

MMIO

- Physical I/O is tied to virtual memory addresses
- Allows us to use the same set of instructions for accessing both memory and I/O devices
- Our favorite use case in lab?
- Volatile keyword
    - Use cases?

MMIO

```
#include <stdio.h>
#include <inttypes.h>
#define REG_FOO 0x40000140
int main(void) {
        volatile uint32_t *reg = (uint32_t *)(REG_FOO);
        *reg += 3;
        print_uint(*reg);
        return 0;
}
```

Example from lecture slides :)

MMIO

- Load instruction.
  - A bus read operation commences.
  - The CPU drives the address "reg" onto the address bus.
  - The CPU indicated a read operation is in process (e.g., R/W#).
  - Some "handshaking" occurs.
  - The target drives the contents of "reg" onto the data lines.
  - The contents of "reg" are loaded into a CPU register (e.g., r0).
- Add instruction.
  - An immediate add (e.g., add r0, #3) adds three to this value.
- Store instruction.
  - A bus write operation commences.
  - The CPU drives the address "reg" onto the address bus.
  - The CPU indicated a write operation is in process (e.g., R/W#).
  - The CPU drives the contents of "r0" onto the data lines.
  - Some "handshaking" occurs.
  - The target stores the data value into address "reg".

Advanced Peripheral Bus (APB)

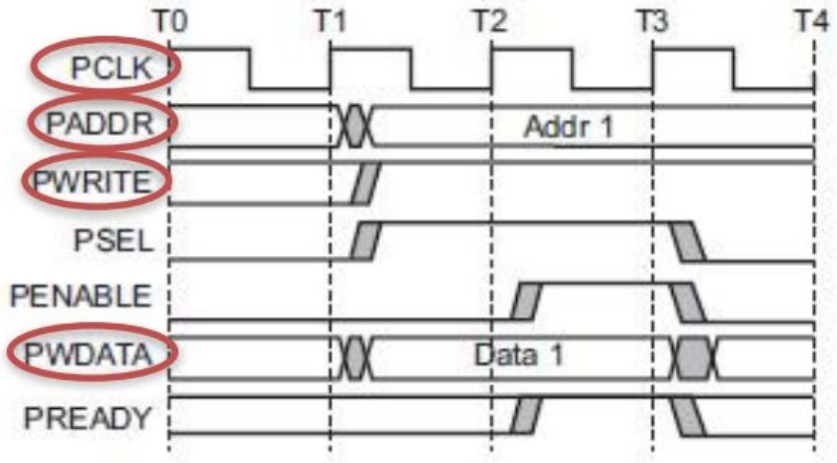
PCLK (Shared)
- Clock

PADDR (Shared)
- Address on bus

PWRITE (Shared)
- 1=Write, 0=Read

PWDATA (Shared)
- Data from processor

PRDATA (Unshared)
- Data to processor



*Note the separate data lanes for read and write

Advanced Peripheral Bus (APB)

PSEL (Unique to target device)

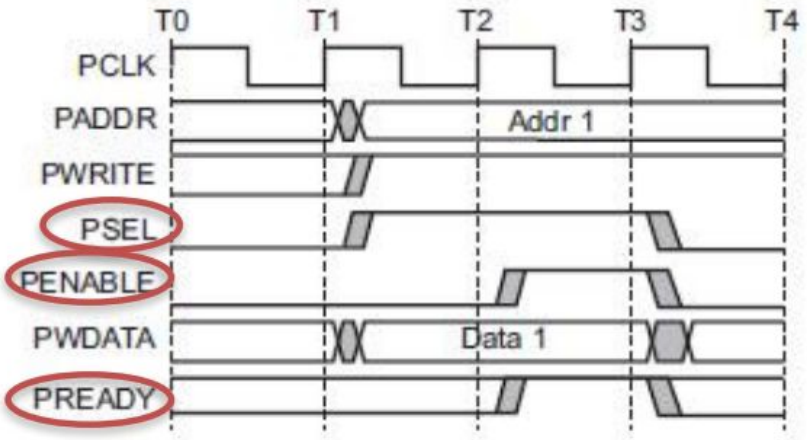
- Asserted if the current bus transaction is targeted to this device.

PENABLE (Shared)

- High during entire transaction other than the first cycle. Distinguishes between idle, setup, and ready.

PREADY (Unique to target device)

- Driven by target. Similar to #ACK. Means target is ready.



Build Process

Compiler

- Volatile keyword
  - Tells compiler not to optimize out references to variable

Linker combines user and C library .o files

- Weak function declarations
  - User can override function declaration
- External declarations
  - Clarifies variable type declarations
  - Allows use of external function

Interrupts

- Exceptions caused by hardware / external events
  - Allows for event-driven program, better alternative to polling
- Interrupt Service Routines (ISR)
  - Uses interrupt vector (ARM) or jump table
- Can be disabled or prioritized
  - Critical sections
  - Priorities
- Shouldn’t allow time-critical sections of code to be interrupted
  - How do we accomplish this?
- Shared data?

Timers

- Better than software loop-based delays
- Allow for more precise timing, delays, and control
- Two basic uses for timers:
  - Measure how long something takes - “Capture”
  - Have something happen once or every X time period - “Compare”
- Virtual Timers
  - We can make one hardware timer look like multiple timers through software
- PWM
  - Prescaler, CCR, ARR

Debugging

Control Complexity

- Control the number of individual components in the system
- Control the number of dependencies in the system
- Incrementally add complexity to the system

Troubleshooting

- Start with possible hardware issues and then move onto software
- Choose the right simplest solution to the problem (sometimes hardware is easier, sometimes software is easier)

Questions?

- Otherwise, will move into actual questions

Assembly - Midterm 1A W14

4) Write an ARM assembly language procedure that implements the following C function in an EABI-compliant manner and conforms to the following signature. *Clearly comment your code* so we can figure out what we are doing and what value each register holds. Poorly commented/unclear code will get points removed. You are to assume "printit()" is an ABI-compliant function that already exists. **[18 points]**

```
uint32 Thing1(uint32 erf, uint32 x[])
{
      int y;

      printit(x[erf]);

      if(x[erf]>erf)
            y=x[erf];
      else
            y=x[1];
      return(y+1);
}
```

r0 r1 r0

```
Thing1:
  push{LR, r4-r6}
  move r4, r0 // let r4 = erf
  move r5, r1 // let r5 = x[]
  LSL r0, #2 // erf x4
  LDR r0, [r1, r0] // Load from r1+4erf.
  push{r0}
  BL printit // Branch to printit.
  pop{r0}.
  cmp r0, r4 // x[erf] > erf
  BLE else.
if: move r6 r0
  B endif
else: LDR r6, [r5, #4]
endif: ADD R0 r6, #1
  POP{PC, r4-r6} bx Lr
```

Assembly - Midterm 1A W14

```
uint32 Thing1(uint32 erf, uint32 x[])
{
      int y;

      printit(x[erf]);

      if(x[erf]>erf)
            y=x[erf];
      else
            y=x[1];
      return(y+1);
}
```

```
Thing1:
      mov r2, r0
      ldr r0, [r1, r2, lsl #2]
      push{r0, r1, lr}
      bl printit
      pop{r0, r1, lr}
      cmp r0, r2
      ble else
if:
      mov r3, r0
      b endif
else:
      ldr r3, [r1,#4]
endif:
      mov r0, r3
      add r0, #1
      bx lr
```

Pointer exercise - F17

6. **(10 pts.)** Pointer Exercise
Provide the values of the memory locations after this code executes. Assume the initial memory values are zero. Express memory contents in HEX. Accesses are little endian.

```
mov r0, #100
mov r1, 0x12
movt r1, 0xef
movw r2, 0xabcd
str r2, [r0],0
str r1, [r0,2]!
strh r1,[r0,-1]
strb r2,[r0]
```

The following list of addresses are in decimal.

- 100:
- 101:
- 102:
- 103:
- 104:
- 105:
- 106:
- 107:
- 108:

Pointer exercise - F17

6. **(10 pts.)** Pointer Exercise
Provide the values of the memory locations after this code executes. Assume the initial memory values are zero. Express memory contents in HEX. Accesses are little endian.

```
mov r0, #100
mov r1, 0x12
movt r1, 0xef
movw r2, 0xabcd
str r2, [r0],0
str r1, [r0,2]!
strh r1,[r0,-1]
strb r2,[r0]
```

The following list of addresses are in decimal.

- 100: CD
- 101: 12
- 102: CD
- 103: 00
- 104: EF
- 105: 00
- 106: 00
- 107: 00
- 108: 00

MMIO - F17

3. **(10 pts.)** MMIO
Write a C function that takes an integer input, compares the input to a threshold, and turns on a LED if and only if the input is greater than the threshold. The LED is mapped to memory address 0x45001234 and has two states: on and off. The threshold value is an integer with range [0:255] and can be read from the register mapped to memory address 0x4500FFFF. The LED is active-high.

```
void LEDoutput(int val) {
  volatile uint32_t * threshold_reg = (uint32_t *)(0x4500ffff);
  volatile uint8_t * led_reg = (uint8_t *)(0x45001234);

  if(val > *threshold_reg) {
    *led_reg = 1;
  } else {
    *led_reg = 0;
  }
}
```

Interrupts - Midterm 1C F17

(7 pts.) You have designed a robot controller with three external interrupts.

- Interrupt A occurs when the robot detects it will soon collide with an object. It is in the highest priority group (priority group 1) and has a subpriority of 2.
- Interrupt B occurs when the robot's battery energy is low. It is also in the highest priority group (priority group 1) and has a subpriority of 3.
- Interrupt C occurs when the robot enters a power saving mode. It is in priority group 2 and has a subpriority of 1.

For each of the following situations, indicate the order in which interrupts are handled by the processor, using "A", "B", and "C". **Assuming "handle" means "enter ISR".**

i. All of the interrupts occur at the same time.
ii. Interrupt C is executing when interrupts A and B trigger at the same time.
iii. Interrupt B is executing when interrupts A and C trigger at the same time

Interrupts - Midterm 1C F17

(7 pts.) You have designed a robot controller with three external interrupts.

- Interrupt A occurs when the robot detects it will soon collide with an object. It is in the highest priority group (priority group 1) and has a subpriority of 2.
- Interrupt B occurs when the robot's battery energy is low. It is also in the highest priority group (priority group 1) and has a subpriority of 3.
- Interrupt C occurs when the robot enters a power saving mode. It is in priority group 2 and has a subpriority of 1.

For each of the following situations, indicate the order in which interrupts are handled by the processor, using "A", "B", and "C". **Assuming "handle" means "enter ISR".**

i. All of the interrupts occur at the same time. **A, B, C**

ii. Interrupt C is executing when interrupts A and B trigger at the same time. **A, B, C**

iii. Interrupt B is executing when interrupts A and C trigger at the same time **B, A, C**

MMIO - F17

3. **(10 pts.)** MMIO
Write a C function that takes an integer input, compares the input to a threshold, and turns on a LED if and only if the input is greater than the threshold. The LED is mapped to memory address 0x45001234 and has two states: on and off. The threshold value is an integer with range [0:255] and can be read from the register mapped to memory address 0x4500FFFF. The LED is active-high.

```
void LEDoutput(int val) {
```

Timers and MMIO - Midterm 1B W17

7) (13 pts.) Your task is to output a 100 Hz PWM signal by writing 0x1 and 0x0 to GPIO 1, which is memory-mapped to at address 0x12345678. Your code must be written in C, and entirely implemented in an interrupt service routine (ISR) of a count-up timer with a 10 MHz clock (named Timer_ISR) that has both a timer compare register and timer overflow register, like that in Lab 5. The timer is already configured to fire interrupts on both compare and overflow events, and a status register (where the 0th bit signifies a compare event and the 1st bit signifies an overflow event) is located at 0x876543218. The timer compare register can be found at 0x87654320, and the timer overflow register can be found at 0x876543214.

At the end of each PWM period (every 1/100th of a second), please check register 0x45671234 for a duty cycle percentage (as an integer from 0 to 100). You do not have to worry about values outside of that range. If the duty cycle percentage changes, you should immediately change to outputting the new duty cycle percentage.

Register table map:
0x12345678: GPIO 1 (memory-mapped IO, PWM output)
0x45671234: PWM duty cycle percentage
0x876543210: Timer compare value
0x876543214: Timer overflow value
0x876543218: Timer interrupt status (If 0th bit is a 1, then the interrupt was a compare interrupt. if 1st bit is 1, then an overflow interrupt. They are guaranteed to not conflict.)

```
// triggers whenever the 10 MHz timer reaches the compare value
void Timer_ISR ( void ) {
```

Timers and MMIO - Midterm 1B W17

7) (13 pts.) Your task is to output a 100 Hz PWM signal by writing 0x1 and 0x0 to GPIO 1, which is memory-mapped to at address 0x12345678. Your code must be written in C, and entirely implemented in an interrupt service routine (ISR) of a count-up timer with a 10 MHz clock (named Timer_ISR) that has both a timer compare register and timer overflow register, like that in Lab 5. The timer is already configured to fire interrupts on both compare and overflow events, and a status register (where the 0th bit signifies a compare event and the 1st bit signifies an overflow event) is located at 0x876543218. The timer compare register can be found at 0x87654320, and the timer overflow register can be found at 0x876543214.

At the end of each PWM period (every 1/100th of a second), please check register 0x45671234 for a duty cycle percentage (as an integer from 0 to 100). You do not have to worry about values outside of that range. If the duty cycle percentage changes, you should immediately change to outputting the new duty cycle percentage.

Register table map:
0x12345678: GPIO 1 (memory-mapped IO, PWM output)
0x45671234: PWM duty cycle percentage
0x876543210: Timer compare value
0x876543214: Timer overflow value
0x876543218: Timer interrupt status (If 0th bit is a 1, then the interrupt was a compare interrupt. if 1st bit is 1, then an overflow interrupt. They are guaranteed to not conflict.)

// triggers whenever the 10 MHz timer reaches the compare value
void Timer_ISR (void) {

```
void Timer_ISR(void) {
  uint32_t * GPIO1 =    (uint32_t *)  0x12345678;
  uint32_t * compare =  (uint32_t *)  0x876543210;
  uint32_t * overflow = (uint32_t *)  0x876543214;
  uint32_t * status =   (uint32_t *)  0x876543218;
  volatile uint32_t * duty_cycle = (uint32_t *) 0x45671234;

  // set overflow
  *overflow = 10000000/100; // aka 100,000

  // set output
  if(*status & 0x1) {
    *GPIO1 = 0x0;
  }
  else if (*status & 0x2) {
    *GPIO1 = 0x1;
    *compare = duty_cycle / 100.0 * 10000000/10;
  }
}
```